Artificial Intelligence Will Never Match Human Intelligence

Baruch Youssin
quququitty@yahoo.com

April 30, 2025

Abstract

We show that the artificial intelligence as we know it now, can never match the human one.

We argue that if artificial intelligence algorithm created by humans is capable of doing all tasks humans can do, then it is an ultimate artificial intelligence algorithm and all such algorithms are equivalent in their capabilities.

If such algorithm exists, we arrive to a contradiction by building a mathematical statement whose truth humans can see but this algorithm cannot.

Contents

1 Main

Recent impressing advances in AI have created a widespread impression that superhuman AI is on the horizon; it has even received a name *superintelligence*, and the anticipated moment it appears, has received a name: *AI Singularity*.

The purpose of this paper is to show that this is not going to happen: algorithms as we know them, will never exceed humans in building new algorithms and in proving mathematical (arithmetical) theorems.

We shall see that the difficulty algorithms have with arithmetical theorems, is to be able to validate the truth of a certain statement informally in cases it cannot be done formally.

For many, this result would not come as a surprise, since AI models only study well the existing literature on which they are trained; they cannot show any *creativity* beyond that — they are *donkeys carrying books* (Koran 62:5). Our result shows this in a formal way.

Our methods use mathematical logic: Church-Turing thesis, enumerability, Tarsky's theorem, MRDP theorem.

2 Results

2.1 Algorithms

2.1.1 The notion

AI as we know it, is an algorithm, and this notion is well-known: it includes all kinds of computer programs, and abstract mathematical notions, of which we shall mention *Turing machines* (these are abstract versions of computers having memory, see [2], Ch. 3) and *recursive functions* which will be used in our arguments (these are mathematical manipulations on natural numbers; for the definition, see [1], Ch. 5).

Algorithms may have one or more inputs:

- Computer programs take sequences of bits (0 or 1), of varying length. This includes all kinds of digital objects: numbers, texts, images, sounds, movies, etc.
- Recursive functions take inputs as natural numbers; these are equivalent to sequences of bits since such sequences may be counted and thus replaced by their numbers. (These numbers may be different from the numbers represented by the sequences, to distinguish between different quantities of leading zeroes.)
- The number of inputs of an algorithm may be fixed or varying. In particular, any recursive function takes a fixed number of inputs while a computer program may take a varying number of inputs. A Turing machine takes a sequence of symbols of varying length; if the symbols are 1 and empty space, this may be interpreted as a sequence of natural numbers of varying length, the numbers being the lenghts of consecutive blocks of symbols 1; see [3], §67.

Algorithms may yield a result which may be one bit (0 or 1, False or True), or a sequence of bits. More generally, a computer program may yield a variable number of outputs, which can be digital objects, similarly to inputs. Similarly, recursive functions yield one or more natural numbers.

Computer programs (and Turing machines) may finish in finite time or not; similarly, recursive functions may yield an answer or not. (Recursive functions that do not always yield a result, are called *partial recursive*; recursive functions that always yield a result, are called *general recursive*.)

Church-Turing thesis states that all these notions are essentially equivalent: if one of them can accomplish some task, then all the others can.

This statement can be proved for any two formal notions of algorithm. It is generally accepted that Turing machines are equivalent to modern computers and most modern programming languages (some programming languages may be weaker than Turing machines). For the proof of equivalence between Turing machines and partially recursive functions, see [3], §68.

2.1.2 Hardware

Our notion of algorithm may include some hardware requirements but not a specific hardware to be used; we do not distinguish algorithms that perform the same manipulations on the inputs but using different hardware.

We shall assume that our algorithms have the most powerful hardware available at the time. In case there are different competing hardware configurations, we shall assume that they are connected and the algorithm is able to use all of them.

2.1.3 Algorithms creating and executing other algorithms

Modern AI algorithms have the possibility of executing other algorithms they create, and we shall consider only the algorithms that have this property: if an

algorithm creates another algorithm, than it is able to execute it. (Hardware is not a problem by our assumptions, even if the created algorithm has more stringent requirements than the original one.)

2.1.4 Randomality

Many modern AI algorithms behave nondeterministically; they achieve it by using some random numbers at certain points, and we shall consider such random numbers as additional inputs.

In our analysis of such algorithms, we fix the value of these random inputs; in other words, we consider algorithms with the different random inputs as different algorithms. In this way we reduce the statements on algorithms with randomality to statements that concern deterministic algorithms.

(Note that this notion of randomality is different from nondeterminism described in [2], Section 1.2, as the algorithms described there, branch at each randomality point into different copies that receive different random inputs; modern AI algorithms do not do that. The nondeterministic algorithms described in loc. cit., are equivalent to the deterministic ones, as shown in loc. cit.)

2.1.5 Time limits

Some of modern AI algorithms yield better results if given more time. This can possibly be achieved by two ways:

- An algorithm can have different versions (an additional input that specifies such version, faster or slower).
- An algorithm can check the time at certain points, and curtail some of its work if the time is running out. This can be also described by introducing the times at the checkpoints as additional inputs. Note that such algorithm may be theoretically unlimited in time - infinite; however, the point of checking time is to finish within certain time bounds, and this is achieved by running only finite versions.

In both cases different values of these additional inputs specify different algorithms, each taking finite time, and our results apply to each of them.

2.1.6 Quantum algorithms

It is generally accepted that the algorithms for quantum computers [4] in their different flavors [5] satisfy Church-Turing thesis, and are thus covered by our results; their advantage in speed is not a concern for us.

2.1.7 Coding

We have mentioned that algorithms may take inputs as digital objects, and produce outputs as such objects, and this is achieved by representing these objects as natural numbers or sequences of bits.

We formalize this by assigning digital codes to the objects.

Given any digital object O (number, text, image, sound, movie, algorithm, etc.), we associate a code to it. We shall denote this code by code(O); it is a sequence of bits. Simply speaking, code(O) is the digital representation of O with all structure removed. For any object type, there could be different ways to define the codes, and we shall fix one of them. In case we need the code to be an integer, we prepend 1 to this sequence of bits and take the integer represented by the resulting sequence.

Given a code c and an object type, we should be able to reconstruct the object O easily; this is achieved by defining the codes in a straightforward way, depending on the object type, and allowing for the possibility that there is no object O of the specified type such that code(O) $= c$ in case c is an illegitimate code for this object type.

2.2 Algorithms matching humans

2.2.1 Such algorithms are equivalent

The purpose of this paper is to show the following informal statement: no algorithm as above can match human intelligence.

We shall assume that such algorithm exists, and arrive to a contradiction.

An algorithm that matches human intelligence should be able to do anything that any human can do, in reasonable time that should not exceed the time needed by a human to do the same thing.

One of the things that humans do, is creating algorithms; thus, an algorithm that matches human intelligence should be able to create any algorithm that humans can create.

On the other hand, if one algorithm creates another one, then the first one can also execute the second one (see Section 2.1.3).

Hence, the results of the second one can be viewed as the results of the first one; the second one is just a part of the first one.

It follows that if algorithms that match human intelligence, exist and can be created by humans, then any of them can create and execute any other one and achieve its results.

In other words, all of such algorithms are essentially equivalent: whatever one can do, any other one can do too.

2.2.2 The ultimate algorithm

In other words, there is the ultimate algorithm that humans can create, having different but equivalent versions; we shall denote any such version by U.

It can do whatever any human can do. Moreover:

> Any such U can do any task that any human-created algorithm (not necessarily the ultimate one) can do. (1)

U takes a variable number of inputs $I_1, I_2, ..., I_n$ that can have any meaning as in Section 2.1.1, and returns also a variable number of outputs:

$$(J_1, ..., J_m) = U(I_1, ..., I_n) \ ;$$

U can do anything that any human can do, in reasonable time.

2.2.3 Our main result

Theorem 1. *The ultimate algorithm does not exist.*

We assume U exists, and arrive to a contradiction.

2.2.4 Solving various tasks

U should be able to solve tasks that humans can solve. Any such task takes its specific inputs $I_t, ..., I_n$.

The ultimate algorithm U may also need to know the task description and the meaning of the inputs; these are additional inputs, and we assume that they are encoded in the inputs of U that go first, $I_1, ..., I_{t-1}$ (the *prefix*), so that the result of the task is $U(I_1, ..., I_{t-1}, I_t, ..., I_n)$.

2.2.5 Human progress

What humans can do and achive, depends on time, and the ultimate algorithm may depend on this time: whenever we say "whatever humans can do", we refer to a certain time. We shall show that the ultimate algorithm does not exist whatever time we choose, but not before the time this paper has been written.

2.3 Arithmetical statements

An arithmetical statement is a mathematical statement about natural numbers, possibly a general one ("for any natural number such that ..." or "there exists a natural number such that...", and such phrases may be repeated multiple times and combined); this statement can be either true or false, and it is possible that establishing its truth may take infinite time, unless we find some general proof.

Such statement may have one or more arguments (also called *free variables*); they are similar to the inputs of an algorithm and the statement becomes true or false if these arguments are given some natural values. A simple example is the statement $x > 3$; it is false if the argument x is given values 1, 2 and 3, and true otherwise.

There are different ways to make this notion precise; the most popular one is a *Formula in Peano arithmetic* (a *Peano formula*); for the precise definition, we refer to [1], 1.2.

An alternative way that we shall use in our proof, is a formula in *Smullian language of arithmetic*, see [1], 2.10; its advantage is the possibility of self-reference, see below.

These two notions of arithmetical statement are equivalent; for the proof, see [1], Ch. 2, Proposition 10.4.

Each of these notions of an arithmetical statement is a text in a certain language; as such, it can be one of the inputs of our algorithms. In particular, it has a code, code(F); given a code c, the formula F_c that has this code, can be reconstructed uniquely and efficiently so that code(F_c) = c, with the possibility that F_c = *invalid* in case the code c is illegitimate.

2.4 The ultimate arithmetician

The ultimate algorithm U should be able to prove all arithmetical statements without arguments that humans can prove, and possibly more; it should not make mistakes claiming that it proved a statement that is actually false.

We are not interested in the details of the proofs created by U but rather in correctly determining that an arithmetical statement is true.

Determining the truth of an arithmetical statement F (a Smullyan formula) is a human task as in Section 2.2.4; it can be performed by U using its task description $(I_1, ..., I_{s-1})$.

In such way we get the *ultimate arithmetician*:

$$U_a(F) = U(I_1, ..., I_{s-1}, I_s = F) \in \{\text{True}, \text{False}\} \ .$$

It has the following properties:

$$U_a(F) = \text{True if humans can prove that } F \text{ is true within reasonable time,} \tag{2}$$

and

$$\text{if } U_a(F) = \text{True then } F \text{ is actually true} \tag{3}$$

("U_a is not making mistakes").

As F may be either true or false, property (3) is equivalent to the following:

$$U_a(F) = \text{False if } F \text{ is actually false.} \tag{4}$$

2.5 The confusing arithmetical statement

Given U, there is an arithmetical statement C without arguments which is equivalent to the following:

$$\text{This statement is not determined as true by the ultimate arithmetician } U_a \ . \tag{5}$$

We shall show that this statement C confuses U_a.

We note that C refers to itself, and the proof of Theorem 1 is based on that; it is an example of self-reference that leads to the well-known liar paradox (*This statement is not true*: this statement cannot be true or false and hence, is not an arithmetical statement; this shows the difficulty of defining arithmetical statements with self-reference).

The statement C here is an arithmetical statement without arguments and can be either true or false; we construct it in Section 4.2.

2.6 Proof of Theorem 1

Suppose C is false. Then, by (5), C is determined as true by the ultimate arithmetician U_a: $U_a(C) =$ True. By (3), C is true, a contradiction to our assumption that it is false.

As C can be either true or false, and it cannot be false, we have determined that C is true. Then, by (5), C is not determined as true by the ultimate arithmetician U_a: $U_a(C) =$ False. Thus, we have determined that C is true but the ultimate arithmetician U_a cannot; we are better!

It follows that we have found a statement that the ultimate arithmetician U_a cannot prove but we can! This is a contradiction that shows that the ultimate algorithm U does exist.

(The human author notes here that he has possibly not convinced the reader that he has written this paper himself and did not use any existing AI algorithm, even though he claims this is the case. Even if the reader admits the possibility that this paper was written by an algorithm, the conclusion is that this algorithm has proved that C is true while U_a has not. This also contradicts the assumption that U is ultimate, see (1).)

The proof is complete.

3 Discussion

3.1 Tasks that challenge AI

Our arguments show that there are two kinds of tasks that humans do better than AI:

- Creating new AI algorithms. The profession of data scientist/machine learning expert is not going to be replaced by AI.
- We have seen that humans are better in validating mathematical statements creatively when the formal validation is impossible. I think this is a part of a more general pattern: humans are better in validating correctness and appropriateness of various things in an informal and substantial way (rather than checking the formal requirements). The profession of substantial/informal validators/QA experts (and possibly other professions with similar duties) is not going to be replaced by AI.

Nevertheless, AI will keep assisting humans in these professions.

3.2 Understanding

The fact that humans are better in substantive validation beyond the formal one can be phrased differently: humans have informal *understanding* while algorithms are limited by their formal rules.

3.3 Safety

AI has already exceeded humans in many fields, and will exceed in many others.

In particular, our result does not exclude the events depicted in the famous fiction movie *The Matrix* which starts with humans being enslaved by machines. In accordance with our result, the movie correctly shows that humans eventually hack the machines.

Our result does not diminish the need for safety precautions in AI development.

3.4 Algorithms as we know them

Our result refers only to AI algorithms as we know them; it does not exclude the possibility that some future AI based on completely new principles — not on the algorithms as described above — will match and exceed human intelligence. However, this possibility appears unlikely to the author.

4 Methods

Here we provide the construction of the confusing arithmetical statement C of Section 2.5.

Denote $\mathbb{Z}^+ = \{1, 2, 3, ...\}$.

4.1 Algorithms and arithmetical statements

Proposition 1. *Any algorithm A with one natural input x which outputs either 1 or 2 for any value of x, can be represented by an arithmetical statement S with one argument x (which we interperet as either a Peano or a Smullyan formula with one variable) such that $S(x)$ is true if $A(x) = 1$ and false if $A(x) = 2$.*

Proof. This proposition follows from deep results in mathematical logic.

We are given an algorithm A with one natural input x which yields either 1 or 2 for any value of the input. By Church-Turing thesis, it is implemented by a general recursive function with one variable, which we shall also denote by A.

Let $T = \{x \in \mathbb{Z}^+ \mid A(x) = 1\}$ be the set of values in $\mathbb{Z}^+$ on which A takes value 1; it is (recursively) enumerable by [1], Ch. 5, Proposition 4.2. (In fact, it is even decidable — [1], Ch. 5, Definition 4.13 — but we shall not need that.)

By MRDP theorem (see [1], Ch. 6, Theorem 1.2), T is Diophantine, and hence, its characteristic function

$$\chi_T(x) = \begin{cases} \text{True if } x \in T \\ \text{False otherwise} \end{cases}$$

can be represented by a Peano or Smullyan formula S; $S(x)$ is true if $A(x) = 1$ and $S(x)$ is false if $A(x) = 2$. □

4.2 Formal construction of the confusing arithmetical statement

We are given the ultimate arithmetician algorithm U_a as in Section 2.4.

Construct another algorithm U_{ac} with one natural input and one natural output,

$$U_{ac}(c) = \begin{cases} 1 \text{ if } U_a(F_c) = \text{ True}, \\ 2 \text{ if } U_a(F_c) = \text{False or } F_c = \textit{invalid} \end{cases}$$

(recall that F_c is the Smullyan formula whose code is the integer c or *invalid*; we shall assume that the codes of Smullyan's formulas are their *numbers*, defined in [1], Chapter 2, 11.1, which allow for easy algorithmic reconstructing the formulas from them).

By Proposition 1, U_{ac} can be represented by a Smullyan formula S with one argument, so that $S(c)$ is true if and only if $U_{ac}(c) = 1$.

Tarsky's theorem says that the truth of Smullyan formulas could not be a determined by a Smullyan formula, i.e., there could be no Smullyan formula S' such that $S'(c)$ is true if and only if F_c is true.

This statement is close to what we need, and we extend the ideas of the proof of Tarsky's theorem to our case, as follows.

Given any Smullyan formula P with one free variable, the proof of Tarski's theorem (see [1], Chapter 2, Theorem 11.4) constructed another Smullyan formula C without free variables which is true if and only if $P(\text{code}(C))$ is false (i.e., C says: "I do not satisfy P").

Taking $P = S$ yields the Smullyan formula C which says: "I do not satisfy S", i.e., "$S(\text{code}(C))$ is false", i.e., "$U_{ac}(\text{code}(C)) = 2$", i.e, "$U_a(C)$ is false".

In other words, C says: "This statement is not determined as true by the ultimate arithmetician U_a".

Thus, C is the required confusing arithmetical statement.

References

[1] Yu. I Manin, *A course in mathematical logic for mathematicians*, 2nd edition, Springer, 2010.

[2] M. Sipser, *Introduction to the Theory of Computation*, 3rd edition, Cengage Learning, 2012.

[3] S. C. Kleene, *Introduction to Metamathematics*, North Holland, 1952.

[4] M. A. Nielsen, I. L. Chuang, *Quantum Computation and Quantum Information*, 10th Anniversary edition, Cambridge University Press, 2010.

[5] M. Mosca, *Quantum Algorithms*, in the volume: Computational Complexity: Theory, Techniques, and Applications, Springer, 2012, p. 2303–2333.